Darpa/Navy Contract No. N00014-92-J-1809

ControlShell: A Real-Time Software Framework

Quarterly Progress Report — April – June, 1993

P.I.'s: Prof. J.C. Latombe, Prof. R.H. Cannon, Jr, Dr. S.A. Schneider

Executive Summary

We are creating a new paradigm for building and maintaining complex real-time software systems for the control of moving mechanical systems. This objective is being met through the *simultaneous* development of both a powerful software environment and cogent motion planning and control capabilities. Our research concentrates on three key areas:

- Building an innovative, powerful real-time software framework,
- Implementing new distributed control architectures for intelligent mechanical systems, and
- Developing distribution architectures and new algorithms for the computationally "hard" motion planning and direction problem.

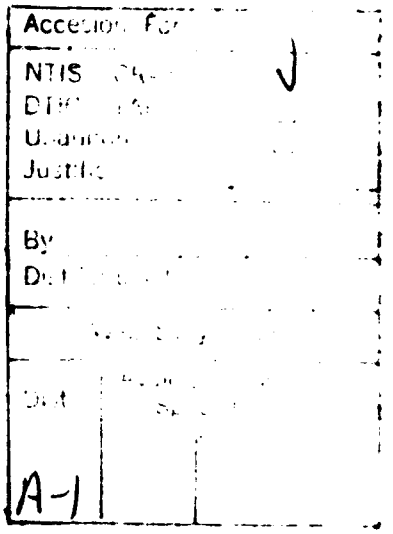
Perhaps more importantly, we are working on the *vertical integration* of these technologies into a powerful, working system. It is only through this coordinated, cooperative approach that a truly revolutionary, usable architecture can result.

Summary of Progress

This section highlights some of our achievements for this quarter. During this period, we have:

- Ported the Network Data Delivery Server to 5 new architectures and added a routing facility.
- Designed and implemented the graphical Finite-State Machine (FSM) Editor to simplify the generation of strategic-level controllers.
- Began the design process for the graphical Data-Flow Editor (DFE) tool for building complex control systems using a block-diagram paradigm.
- Began the design of the a new version of ControlShell for C++ development.

- Developed and implemented the Task Interface.
- Added vision targets to the robot.
- Identified and calibrated the arm-kinematic parameters with respect to the vision system.
- Developed and implemented a controller for the last two degrees of freedom of the robot.
- Built digitally controlled pressure-switching boxes to actuate the pneumatic grippers.
- Developed a new approach to escaping local-minimum during path planning based on the precomputation of a network of collision-free configurations.
- Design and implementation of a randomized three-arm manipulation planner for manipulating an elongated object in a 3D cluttered environment.
- Enhanced robot simulator to incorporate actual robot dynamics for trajectory tracking.



Chapter 1

Introduction

The goal of this research project is to build a new paradigm for building and maintaining complex real-time software systems for the control of moving mechanical systems. This objective is being met through the *simultaneous* development of both a powerful software environment and cogent motion planning and control capabilities. Our research concentrates on three areas:

- Building an innovative, powerful real-time software development environment,
- Implementing a new distributed control architecture, and using it to deftly control and coordinate real mechanical systems, and
- Developing a computation distribution architecture, and using it to build on-line motion planning and direction capabilities.

We believe that no technology can be successful unless proven experimentally. We are thus validating our research by direct application in several disparate, real-world settings.

This concurrent development of system framework, sophisticated motion planning and control software, and real applications insures a high-quality architectural design. It will also embed, in *reusable* components, fundamental new contributions to the science of intelligent motion planning and control systems. Researchers from our three organizations, the Stanford Aerospace Robotics Laboratory (ARL), the Stanford Computer Science Robotics Laboratory (CSRL), and Real-Time Innovations, Inc. (RTI) have teamed to cooperate intimately and directly to achieve this goal. The potential for advanced technology transfer represented by this cooperative, vertically-integrated approach is unprecedented.

Framework Development This research builds on an object-oriented tool set for real-time software system programming known as *ControlShell*. It provides a series of execution and data interchange mechanisms that form a framework for building real-time applications. These mechanisms

are specifically designed to allow a component-based approach to real-time software generation and management. By defining a set of interface specifications for inter-module interaction, ControlShell provides a common platform that is the basis for real-time code exchange and reuse.

Our research is adding fundamental new capabilities, including network-extensible data flow control and a graphical CASE environment.

Distributed Control Architecture This research combines the high-level motion planning component developed by the previous effort with a deft control system for a complex multi-armed robot. The emphasis of this effort is on building interfaces between modules that permit a complex real-time system to run as an interconnected set of distributed modules. To drive this work, we are building a dual-arm cooperative robot system that will be able to respond to high-level user input, create sophisticated motion and task-level plans, and execute them in real time. The system will be able to effect simple assemblies while reacting to changing environmental conditions. It combines a world modelling system, real-time vision, task and path planners, an intuitive graphical user interface, an on-line simulator, and sophisticated control algorithms.

Computation Distribution Architecture This research thrust addresses the issues arising when computationally complex algorithms are embedded in a real-time framework. To illustrate these issues we are considering two particular problem domains: *object manipulation by autonomous multi-arm robots* and *navigation of multiple autonomous mobile robots in an incompletely known environment.* These two problems raise a number of generic issues directly related to the general theme of our research: motion planning is provably a computationally hard problem and its outcomes, motion plans, are executed in a dynamic world where various sorts of contingencies may exist.

The ultimate goals of our investigation are to both provide real-time controllers with on-line motion reactive planning capabilities and to build experimental robotic systems demonstrating such capabilities. Moreover, in accomplishing this goal, we expect to identify general guidelines for embedding a capability requiring provably complex computations into a real-time framework.

Chapter 2

ControlShell Framework Development

This section describes our progress in developing the ControlShell framework and underlying architecture. Two fundamental extensions to ControlShell are being pursued:

- Distributed information sharing paradigms, by Gerardo Pardo-Castellote and Stan Schneider.
- Graphical Computer Aided Software Engineering (CASE) environments, by Stan Schneider and Vince Chen.

2.1 Distributed Information Sharing Paradigms: NDDS

Aside from the experiments under this contract, a pre-release version of NDDS is being used by other researchers for their independently-funded experiments. These users include the development of an underwater semi-autonomous robotic vehicle in collaboration with the Monterey Bay Aquarium Research Institute (MBARI) and a high level interface to a semi-autonomous free-flying robot at the Aerospace Robotics Laboratory.

NDDS already supported the following platforms: Sun Sparc-Stations with Solaris 1.xx, DEC MIPS-based workstation with Ultrix 4.xx and VME boards with Motorola 680XX processors with VxWorks 5.0.XX as operating system.

Responding to the demands of our users, during this quarter we have ported NDDS to the following additional platforms: Sun Sparc-Stations with Solaris 2.xx, DEC ALPHA-based workstations with OSF1 operating system, HP workstations with HPUX 4.xx and VxWorks 5.1.XX for Motorola and Sparc-based boards.

In order to be successful in the marketplace NDDS must meet the needs of a variety of potential

customers. Although not directly required by the experiments funded under this contract is has become clear through our interaction with current ControlShell customers and other people in the field that several extensions are necessary.

- Reliable updates. NDDS uses UDP/IP to send updates from producers to consumers. UDP is unreliable and un-acknowledged. This means that packets may arrive out-of-order or even be lost. For some applications this may be un-acceptable.

- Distributed query support. Currently NDDS supports subscriptions as the only model for data exchange. While this is an appropriate model for sensor-like data that is constantly changing, there are many applications for which a significant portion of the data rarely changes or is needed very sporadically. These cases will require a distributed-query facility.

- Network routing. In order to service subscriptions NDDS maintains a distributed database of the data being required by each node. This database is maintained using point-to point messages between all the hosts involved in the particular session. Although less efficient than broadcasting, point-to point messages were chosen because they are readily extensible outside the LAN environment. Also point-to-point messages waste bandwidth when most of the communication occurs between hosts connected by a WAN. Two possible solutions are possible: First we could support broadcasting within a LAN. Second we could add a proxy facility so that all the messages between hosts within the same LAN which are addressed to hosts in a different LAN get bundled into a single point-to-point message to a "proxy" server in the remote LAN that will distribute it within the LAN.

  We have augmented NDDS with routing information so that packets can be forwarded over different networks. The original motivation for this change is that often may real-time processors are installed in local sub-nets without an "official" IP address. Therefore, normal IP routing is unable to find these processors from outside the Local Area Network. This is the case for the real-time processors that control the two-armed robot. We plan to add a "proxy" facility to NDDS so that customized "proxy" servers can forward packets across networks.

We will be addressing these issues during the next quarters.

2.2 ControlShell CASE Environment

Significant progress was made in this quarter in the area of developing ControlShell's graphical CASE environment.

ControlShell's event-driven finite state machine (FSM) facility provides easy strategic control. The state machine model features rule-based transition conditions, true callable sub-chain hierarchies, task synchronization and event management. However, it has been difficult to use, as state programs can become fairly complex.

This quarter, we completed an initial implementation of a graphical FSM editor. The editor allows the user to create FSM diagrams in a graphical environment. The FSM Editor generates state machine specification files that the ControlShell run-time parser uses to build FSMs on the real-time machine. These files specify transition modules (routines plus data) to be executed when state transitions occur. The run-time state machines are created on-the-fly without any recompilation of code.

The FSM Editor also generates header files that enumerate transition-routine return codes and stimulus expressions. Users include these headers in their transition-routine code.

The graphical FSM Editor is written entirely in C++, using the MOTIF/X libraries. This ensures that the editor will run on virtually any workstation platform. A preliminary user manual for the FSM Editor is attached.

2.2.1 FSM Overview

The Finite State Machine (FSM) module is designed to provide a simple strategic-level programming structure that also assists in managing events and concurrency in the system. The FSM module combines a non-sequential programming environment with natural event-driven process management. With this structure, the programmer is actively encouraged to divide the problem into small, independently executing processes.

To utilize the FSM module, the programmer first describes the task as a state transition graph. The graph can be directly described within ControlShell's graphical FSM editor (see Figure 2.1). Each transition—represented by an arrow in the graph—specifies a starting state, a boolean relation between stimuli that causes the transition, the CSModule to be executed when the transition occurs, and a series of "return code-next state" pairs that determine the program flow.

The FSM model is quite general; it supports rule-based transition conditions (reducing the number of states in complex systems), true callable sub-chains of states (so libraries of state subroutines can be developed), wild-card matching (so unexpected stimuli can be processed), global matching (allowing easy error processing), and conditional succession (so state programs may easily branch). Transitions are specified as boolean relations of three types of stimuli: transient, latched, and conditional. Transient stimuli have no value, and exist only instantaneously. Latched stimuli also have no value, but persist until some transition expression matches. Condition stimuli have string values; they persist indefinitely and thus represent memory in the system. Thus, the transition condition "Object = Visible AND Acquire" might cause a system to react to an acquisition command from a high-level controller. Providing these three stimuli types allows combination of both "system status" and "event" types of asynchronous inputs into easily-understood programs.

The FSM module takes advantage of the atomic message-passing capability of modern real-time kernels to weave the incoming asynchronous events into a single event stream. Any process can call a simple routine to queue the event; the FSM code spawns a process to execute the resulting event stream. The result is an easy-to-use, yet powerful real-time programming paradigm.

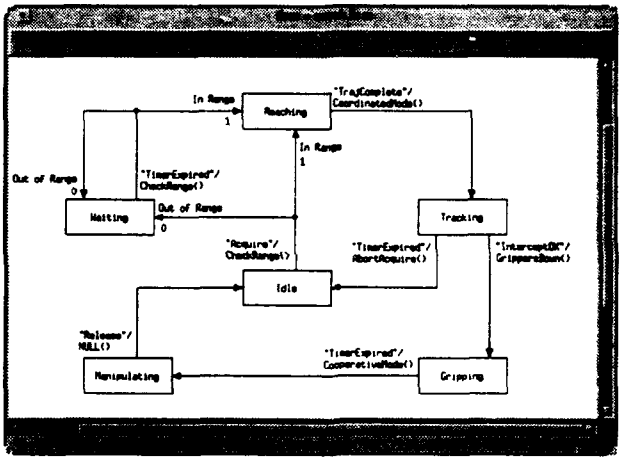


Figure 2.1: **Finite State Machine Editor**

State transition graphs allow easy visualization of multi-step operations; this example is a (simplified) program to catch a moving object with a dual-arm manipulator.

Chapter 3

Distributed Control Architectures and Interfaces

This section covers our research in software architectures, communication protocols and interfaces that will advance the state-of-the-art in the prototyping-development-testing cycle of high-performance distributed control systems. These interfaces will be implemented within the framework described in Chapter 2. The results of this research will be applied to the vertical integration of planning and control and demonstrated by executing a set of challenging tasks on our two-armed robot system.

There are three main thrusts to this research:

- Development of inter-module interfaces for distributed control systems, by Gerardo Pardo-Castellote.
- Development of a control methodology capable of executing high-level commands, by Gerardo Pardo-Castellote, Tsai-Yen Li, and Yotto Koga.
- Hardware development and experimental verification, by Gerardo Pardo-Castellote and Gad Shelef.

This quarter, we devoted considerable attention to the task interface and control of the remaining (z and yaw) degrees-of-freedom of the arms.

3.1 Inter-Module Interfaces

Our system uses three main interfaces to communicate among the four major modules as shown in figure 3.1. All these interfaces use NDDS to provide network connectivity. The world-model

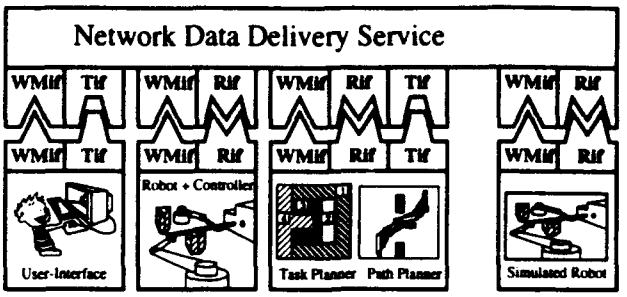


Figure 3.1: **System Architecture**

The overall system showing its four main modules. Each module communicates using one or more of the three interfaces: The World Model Interface (WMif), the Robot Interface (Rif) and the Task Interface (Tif). These modules are physically distributed. The Network Data Delivery Service plays the role of a bus providing the necessary interconnections.

and the robot interfaces where described in April-June and July-September 1992 quarterly reports. This quarter we have implemented the last remaining interface: the Task Interface. The role of the task interface is to communicate user task-commands to the Planner module. The tasks available through this interface are summarized in table 3.1. The abort task-command requires some explanation. In a dynamic environment in which there may be agents that are not under the control of the planner (such as humans or other planners), it becomes necessary to specify what is the "persistence" of a task command. For example, lets say we request a certain object to be placed at a desired location and the system achieves that goal. At some later time, some other agent moves that object away. Should the system try to bring that object back to its initial location? In other words, should the lifetime of that task command issued to the planner extend beyond the time in which it was initially achieved? We feel that the lifetime of the task should in fact extend beyond the time it is achieved for the first time. In a sense this is a natural extension to the task-level of the lower-level control approach. A control system that is commanded to regulate to a certain desired state, is expected to keep the system at that state despite external disturbances. In our case, we consider the task as a higher-level specification of the desired "state" of the workspace. Other agents, are considered "external disturbances" that the system must compensate for. Task commands such as "place these three objects at those locations" modify the desired "state" for the workspace (which now requires those objects to be at the specified locations). The abort command retracts the last task command.

In addition, we have modified the robot interface by combining both the operational-space and the joint-space move-arm commands into a single command. This command specifies both the via-point path in operational space and the corresponding path in joint space. The significance of this is that now, the responsibility for deciding how to merge this information to control the robot, is left to the robot-controller itself. The advantage of this change is that, by leaving the control

command	*meaning*
put objects	Place an set of objects at desired locations. This specifies a goal location (in the global reference frame) for each object. These objects need to be acquired (from their current locations or the conveyor) and placed autonomously.
put arms	Place the arms at a desired location. Any objects currently grasped must be released. The arm location is specified by giving the desired tool location with respect to the global reference frame.
abort	Abort current task.

Table 3.1: Task-Command primitives available through the Task interface.

decisions to the controller, different control policies can be used in a manner that is transparent to the planner module, so that the planner is isolated from the control implementation.

3.2 Control Methodology Development

During this quarter we have:

- Identified position and orientation of the robot bases with respect to the vision system.
- Developed a controller for the last two degrees of freedom of each arm. The Yaw and Z.
- Modified the trajectory-generation algorithm to allow synchronized trajectories.

The vision system is our only sensor capable of tracking the objects in the workspace. What ultimately matters to acquire an object, is that the tool coordinates obtained from the robot kinematics, correspond as closely as possible with the tool's vision coordinates. For this reason, we have found that the best approach is to minimize the average difference between kinematics and vision over the whole workspace. We have achieved this by placing a vision target at the tool and finding the parameters that minimize this average difference. We have developed an automated approach to achieve this calibration. This is important if the arms need to be moved to a different location (which we have done already a couple of times). The arm is commanded to move to a grid of locations covering all the available workspace while data on the joint angles and vision coordinates is gathered. An off-line minimization procedure identifies the location of

the arm bases, the location of the vision target with respect to the tool-frame and initial angular offsets. The minimization achieves a maximum error of 5.5 mm (4 mm average) between kinematic a.. l vision coordinates. This discrepancy (mostly due to vision aberrations but also influenced by the non-perfect verticality of the robot axes) will be corrected by an on-line adaptive algorithm.

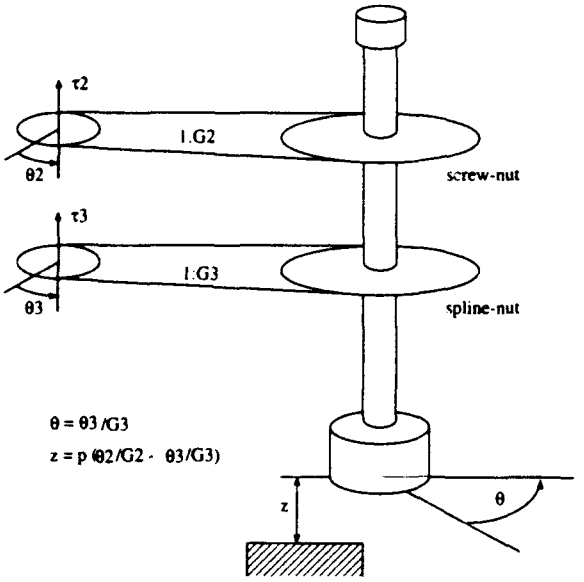


Figure 3.2: **Spline-Screw assembly**

The last 2 degrees-of-freedom of the SCARA manipulators contain a screw-spline assembly. Two nuts drive bearings running in two different grooves carved on the same shaft. The motors are mounted in the elbow and are coupled to the nuts using cable chains.

The last two DOF of our arms, use a spline-screw element that incorporates two DOF (yaw and z) in a single shaft. This shaft contains a ball screw groove and a spline groove. Rotation of the nut containing the bearings that move along the screw groove produce pure Z motion whereas the nut containing bearings along the spline groove produce a coupled Z and yaw motions. This is schematized in figure 3.2. The equations of motion for these two degrees of freedom are:

$$z = p(\theta_2/G_2 - \theta_3/G_3) \tag{3.1}$$

$$\theta = \theta_3/G_3 \tag{3.2}$$

$$\begin{bmatrix} \tau_2 \\ \tau_3 \end{bmatrix} = \begin{bmatrix} I_2/(pG_2) + M_z/G_2 & I_2/G_3 \\ M_z/G_3 & I_3/G_3 \end{bmatrix} \begin{bmatrix} \ddot{z} \\ \ddot{\theta} \end{bmatrix} + gpM_z \begin{bmatrix} 1/G_2 \\ 1/G_3 \end{bmatrix} \tag{3.3}$$

Refer to table 3.2 for definitions of the different symbols.

G_2	Gear-ratio for z-axis motor to screw nut
G_3	Gear-ratio for yaw-motor to spline nut
I_3	Effective inertia of screw nut
I_3	Effective inertia of spline nut
M_z	Effective mass displaced on the Z motion
p	Pitch of the screw
g	Acceleration of gravity

Table 3.2: **Meaning of symbols in equations of motion for last 2 DOF of the arms.**

3.3 Hardware Development and Experiments

This quarter we have made the following developments:

- Added vision targets to the arm end-points to provide the controller with end-point vision information.
- Built digitally controlled pressure-switching boxes to actuate the pneumatic grippers.
- Removed hard calibration-stops from robot bases. Built foam covers for all hard surfaces.

We have performed an initial integration of the controller with the planner. This integration has been successful but system operation wasn't fully reliable. We have identified the following problems which we will address next quarter:

- Power-On calibration. When the system is first brought up, we need an automated calibration procedure for the arm encoders. In the past, hard stops where used. Hard stops reduce the available workspac. nd can damage the arms in the event of a collision (a situation that we have experienced with previous hardware). It was therefore desirable to remove all hard stops, as we did, but this has left the system without a means for accurate power-on calibration.
- Sensor Fusion between kinematic data and vision data. The robot tool contains vision targets, which allow tracking of the endpoint with the same vision sensor that tracks objects in the workspace. The arms are also equipped with joint encoders, which allow the same information to be obtained through the kinematics. The vision data has only a bandwidth of 60 Hz. so it can't be used for the higher-bandwidth arm controllers. If we base our arm control purely on the kinematics, there are places in the workspace where (even after calibration), the difference between kinematics and vision is large enough (up to 5.5 mm), that the arm fails to capture a

(vision sensed) object. We need to develop a sensor-fusion approach to combine kinematic and vision data, so that the new signal has the adequate low and high frequency characteristics.

Chapter 4

On-Line Computation Distribution Architectures

Our research addresses technical issues arising when computationally complex algorithms are embedded in a real-time framework. To illustrate these issues we consider two particular problem domains: *object manipulation by autonomous multi-arm robots* and *navigation of multiple autonomous mobile robots in an incompletely known environment.* These two problems raise a number of generic issues directly related to the general theme of our research: motion planning is provably a computationally hard problem and its outcomes, motion plans, are executed in a dynamic world where various sorts of contingencies may happen.

The ultimate goal of our investigation, concerning the two problem domains mentioned above, is to both provide real-time controllers with on-line motion reactive planning capabilities and build experimental robotic systems demonstrating such capabilities. Moreover, in accomplishing this goal, we expect to elaborate general guidelines for embedding a capability requiring provably complex computations into a real-time framework.

This quarterly report covers work done towards this goal during the period of April, May, and June 1993. During this period, our work addressed on the following areas:

1. Distribution of Path Planning, by Tsai-Yen Li.
2. Parallelization of Path Planning, by Lydia Kavraki.
3. New Methods for Fast Path Planning, by Tsai-Yen Li.
4. Multi-Arm Manipulation Planning in 3D, by Yotto Koga.
5. Experiments in Manipulation Planning, by Tsai-Yen Li and Yotto Koga.
6. Landmark-Based Mobile Robot Navigation, by Anthony Lazanas, Byung-Ju Kang and Ken Tokusei.

7. Mobile Robot Navigation Toolkits, by Craig Becker, Mark Yim and David Zhu.

8. Multi-Mobile Robot Simulator, by Craig Becker and David Zhu.

Areas 1 through 5 are mainly related to the first problem domain, i.e., *object manipulation by autonomous multi-arm robots.*

Areas 6 through 8 are mainly related to the second problem domain, i.e., *navigation of multiple autonomous mobile robots in an incompletely known environment.*

Participating Ph.D. Students: Craig Becker, Lydia Kavraki, Yotto Koga, Anthony Lazanas, Tsai-Yen Li, Mark Yim.

Participating Master Students: Ken Tokusei, Byung-Ju Kang.

Participating Staff: Randall Wilson, David Zhu.

4.1 Distribution of Path Planning

In the previous reports, we have identified several axes to distribute planning:

- distribution over processors,
- distribution over problem approximations,
- distribution over planning methods,
- distribution of commitment over time,
- distribution of problem over time.

We also have implemented a distribution architecture to integrate planning over several single-processor workstations connected by the Local Area Network (LAN).

During this quarter we focus our research on applying these distribution methods to our demonstration scenario: A robotic assembly work station is equipped with two SCARA-type manipulators. The robot arms pick up parts from a conveyor belt and deliver them to their goal locations while avoiding collision with the obstacles in the environment. The parts may require one arm or two arms to deliver them cooperatively but do not need to come with a specific location or orientation. The obstacle locations and goal locations for the parts can be changed dynamically.

Since almost everything in the scenario is dynamic and flexible, the need for on-line planning is clearly illustrated. Due to the complication of the task-level planning and the huge search space it entails, providing a reasonably good on-line solution to this problem becomes more challenging than ever. However, we would like to show that with the distributed planning algorithms and

architecture we are developing, we can provide the best on-line planning service to our robotic system.

For the problem in the above scenario, one can distribute the planning including task planning and motion planning along several axes that we have identified. In this quarter, we investigated the axes of distributing the problem over time and over several processors.

A manipulation task usually consists of several subtasks: catching a moving object, delivering the object, and possibly ungrasping and regrasping the object in order to avoid the joint limits of the links. These subtasks may have different time constraints associated with them. For example, catching a moving object on a convey belt requires the planner to come up with a valid path before the object leaves the arm workspace. Thus, one can decompose the task into several subtasks and distribute the planning for these subtasks over time according to the ordering of these subtasks and their temporal constraints. Under this distribution, the robot can start executing a partial path before a full manipulation path is found and the remaining path can be found in parallel while the robot is moving.

Since we have multiple robot arms in the same work cell, it is often that a given task can be accomplished in several different ways if the object to be manipulated only requires one arm to grasp. We identify these different possibilities and distribute them over several processors. For example, for the object that only requires one arm to grasp, if both arms are available at the moment the planner is invoked, we can distribute the planning over two processors with each arm for each processor. If one processor succeeds and the other fails, compared to the sequential approach (i.e., sequentially trying each arm) the gain of using distribution is obvious. When one process has found a valid path, we can wait for the other process to return an answer (success or failure) and choose the one that has the shortest execution time. However, we can also decide to use the first one that returns a valid path to satisfy certain temporal constraints of the object. Another example of distribution over processors is to consider different grasping postures. Since there can be two possible postures (elbow in and elbow out) for a given arm to grasp an object, we can further distribute the planning for each arm over two processors for each candidate posture.

Currently, we are investigating more possibilities that we can distribute the problem over other axes in order to take advantage of the distribution architecture to solve the problem in an on-line manner.

4.2 Parallelization of Path Planning

During this quarter we focused our research on the critical evaluation of the parallel version of the Randomized Path Planner (RPP), which we have implemented on an SGI 4D/240 (see previous Quarterly Reports). RPP is often a very efficient algorithm and the parallel implementation adds to its efficiency by reducing planning time as described in our previous reports. However, several cases have been identified where RPP behaves poorly. In these cases the use of small-scale parallelism does not seem to improve its performance considerably.

We have noticed that RPP (a potential-field- based method) may fail to find a path in reasonable time when:

- the robot's collision-free space in the configuration space (C-space) consists of several regions (which we call "traps") connected through narrow passages,
- the boundaries of the attraction basins of the potential's local minima are located within or close to those passages, and
- the initial and final configurations lie in two different traps.

Then the potential function cannot help the planner to find a path between two traps. The search inevitably falls in the local minimum of the current trap. RPP attempts to escape this minimum by performing a series of random walks, but the probability that any of these walks finds its way through a narrow passage is almost zero.

One idea to fix this problem is to use several potential functions, hoping that the boundaries of the basins of attraction for one potential will be significantly different from the boundaries for another potential. We have tried this idea, and it works well in some cases. In other cases, it seems very difficult to generate an adequate set of potential functions. Furthermore, each failure with one potential takes a significant amount of time, so that the number of potentials that RPP may consider has to be small.

We have decided to address the "trap problem" in a very different way. This consists of defining a preprocessing of the C-space of the robot (done only once for a given environment), after which path planning from any initial to any final position in that environment will be very fast. During the preprocessing phase we will generate a large set of collision-free configurations (nodes) of the robot and will interconnect them to a network using very simple and fast path planning techniques. We hope to capture the connectivity of the free C-space with our network. Then path planning between any two configurations can be done by connecting both configurations to some two nodes A and B in the network and searching the network for a sequence of edges connecting A and B.

We hope that our ideas will evolve in to a robust planning method which will be able to deal with problems that are difficult to solve with existing path planning techniques. In addition, the proposed approach has a large potential for parallelism in the network construction phase, which we plan to exploit.

4.3 New Methods for Fast Path Planning

One of the goals in this project is to demonstrate the on-line planning capability of our planner in a dynamic environment. For on-line planners, the efficiency is one of the most important issues about the performance of the overall system. We not only need to find a valid path but also need to find it in a reasonably short period of time in order to satisfy possible temporal constraints. A rule of

thumb for a reasonable good on-line motion planner is that the time spent in planning should be only small fractions of the time for executing the generated path. Although the planners we developed in the past are among the fastest planners ever implemented, they do not completely satisfy the on-line requirements of the enhanced robot available in the ARL (Aerospace Robotics Laboratory at Stanford). This quarter, we continue our research in developing more efficient algorithms for on-line applications, more specifically, for the dual-arm robot in the ARL.

The problem in the scenario we describe in the previous section is much more complicated than the manipulation problem we dealt with in the previous versions of planners for the following reasons:

- The search space is much larger since we have two arms and multiple objects in the scene. However, finding a valid path does not always require searching the entire space. How can one reduce the size of the search space and still be able to find a good solution for a given task.

- Since an object may be moving continuously on the conveyer belt, if we cannot find a path to grasp the object in time, it would leave the work space of the arms before the arm can reach the conveyer. Therefore, the efficiency of the planner not only determines the productivity of the robotic system but also determines the feasibility of the tasks to be accomplished.

- The objects move continuously on the conveyer. Assume the speed of the conveyer is constant, how can one predict the future locations of an object and move the arms to grasp the object while accounting for the possible planning and communication delays.

We are developing a new manipulation planner including a task planner and a path planner which takes these on-line constraints into account. The preliminary result is encouraging and will be presented in the next quarter report.

4.4 Multi-Arm Manipulation Planning in 3D

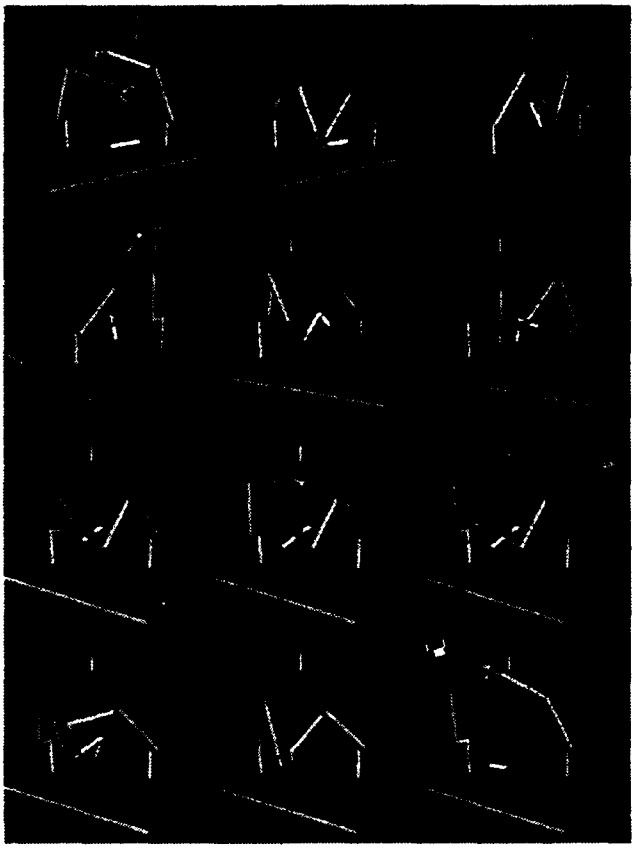


Figure 1. A multi-arm manipulation path. The object requires two arms to move it, but only one arm to hold it statically.

During this quarter we have completed a new randomized manipulation planner to deal with the three-dimensional workspace. We have considered a scenario derived from an actual setting existing in our Computer Science Robotics Laboratory. We have three Puma 560 arms cooperating to manipulate some object to a goal location. As in the planar case, situations arise in the three-dimensional workspace where the arms must ungrasp and then regrasp the object in a new way to achieve the goal. To deal with this manipulation problem we have taken the strategy as outlined in the first quarter report of '93.

An example path found by the planner is shown in Fig. 1. The task is to manipulate the L-shaped object to the other side of the workspace. The object is considered heavy and bulky, consequently

requiring two arms to carry it. Regrasping is achieved by holding the object in a static manner while the arms change their grasp. In this case, we assume that one arm is sufficient to hold the object in a static manner. There are three robot arms in the system and although their kinematic model is based on the Puma 560 arm, the links of the arms are cylindrical in shape to simplify the collision checking. The planned path has the arms carry the object through the opening in the wall and then finally to the goal location. This particular path took approximately one and a half minutes to compute on a DEC alpha workstation.

4.5 Experiments in Manipulation Planning

In this quarter, in close cooperation with ARL, we have completed our preliminary integration of the whole system including a hierarchical robot controller, a world modeller, a simulator, a task planner and several path planners. These software modules run on a real-time operating system and several UNIX based work stations under the Control Shell environment. The system has been tested to be capable of moving a long object which requires two-arm cooperative manipulation. All the user needs to specify is the goal location of the object through a graphical user interface and the motion plan is automatically generated. The task planner also monitors the robot execute the plan to ensure the safe operation of the robot.

We also improved the simulator to emulate the dynamics of the robot. The simulator is designed to provide the full functionality of the real robot. It includes a vision system, a robot controller, and a world modeller and uses exactly the same interface as the robot does. It produces the current location of objects and obstacles in the work space and the current configurations of the arms and broadcasts them to any interested modules in the distributed environment under ControlShell (e.g., the planner, and the user interface). The primitive robot commands it receives from the planner are paths consisting of sets of via points (collision-free configurations) for the robot to follow. The simulator further time-parameterizes these paths according to the dynamic model of the robot and the joint torque constraints that are used in the robot controller. This new feature enable us to verify not only the kinematics of the generated path but also how the robot progressively executes the path under the joint torque limits.

With this simulator, the testing phase of a newly developed planner becomes much easier and feasible when the robot is shut down for rework. Significant amount of time is also saved to integrate the planning and control modules under the Control Shell environment.

4.6 Landmark-Based Mobile Robot Navigation

During the previous quarters we have developed and implemented several efficient algorithms for landmark-based mobile robot navigation. Mobile robot navigation is perhaps the most crucial problem in mobile robotics. Despite a lot of research effort over the past two decades, the problem

still has no satisfactory solution. Prior theoretical studies and experiments with implemented systems tell us that:

- One cannot build a truly reliable system without both making clear assumptions bounding uncertainty and enforcing these assumptions by appropriately engineering the robot and/or its workspace.
- If assumptions are too mild, the planning subproblem is computationally intractable. If assumptions are too strong, engineering is too costly and/or navigation not flexible enough.

Our research investigates the tradeoff between "computational complexity" and "physical complexity" in reliable mobile robot navigation. Our approach consists of:

1. Defining a formal navigation problem with just enough assumptions to make it possible to construct a sound and complete planner that is also computationally efficient.
2. Designing and implementing such a planner, in order to verify that the planner is actually efficient.
3. Engineering a robot and its workspace to enforce the assumptions in the defined problem, in order to verify that the "cost" of such engineering is reasonable.
4. Implementing a navigation algorithm that makes a real robot execute plans generated by the planner, in order to verify that navigation is actually reliable.

This approach also induces a new role for experimentation in robotics: When robot algorithms are proven correct under formal assumptions, the purpose of experimentation shifts from demonstrating that they behave as intuitively expected on a sample of tasks, to verifying that the amount of engineering induced by the assumptions is acceptable.

Our previous quarterly reports describe work on steps 1 and 2 above. This quarter we started dealing with steps 3 and 4. Our work centered on experimentations necessary to implement the landmark-based motion planner developed during the previous quarter. We decided to use visual landmarks that are to be located on the ceiling in an indoor environment to designate the landmark regions, where the mobile robot is assumed to have no uncertainty in sensing. A CCD camera module is installed on the top of the robot, pointing upward to detect and identify landmarks.

Implementation of Vision Function In order to equip the robot with a function to detect landmarks, we implemented a pattern recognition capability. The algorithm is based on computing L_1 norm between pixel-level outlines of a given image and a model (both filtered through an edge detector) and takes $O(mn)$ time, where m, n are the numbers of pixels in the model and the image, respectively.

Landmark Design and Recognition Procedure Using the pattern capability discussed above, we designed landmarks which could provide the following information: relative (x, y) position of the robot, relative orientation of the robot, and the identification of the landmark.

A landmark is a black-on-white symbol which resembles the letter C. It has a 10-inch diameter, an internal cavity ranging from 2 to 7 inches in diameter, and a break that is 1-inch wide. The break gives the landmark an orientation, which is stored in a map along with the location. The diameter of the inner cavity serves to identify the landmark. In order to avoid misidentification, we limited the cavity diameter to be in integral inches between two and seven. Therefore, we currently cannot have more than six distinct landmarks in a given environment.

Recognition of a landmark proceeds as follows. First, after filtering the image with an edge detector, the robot finds the location of the landmark relative to its visual field by looking for a disk whose pixel size matches the perimeter of the landmarks. We assume that the ceiling height is constant, so the image size (in pixels) of the landmark is constant with a possible deviation of ± 2 pixels due to elliptical elongation arising from perspective skew. Also, the break in the landmark introduces a slight mismatch between a full circle and an edge image of the landmark. The algorithm is robust enough to cope with the deformations, allowing the detection to be rotationally invariant.

If a landmark is detected, the center of the disk is recorded as the relative displacement in pixels, which can be easily converted into physical distance (in inches, for example).

The robot proceeds to find the orientation of the landmark by scanning the perimeter of the landmark in circular fashion to find a break. A statistical measure based on histograms determines the intensity threshold to discretize the grayscale image into *white* and *black* pixels. The robot finds the break by looking for *white* pixels.

Finally, the robot identifies the landmark by measuring the size of the internal cavity. The inner diameter is measured by scanning the grayscale image from the center of the landmark symbol toward outside along several radial lines and looking for the *black* pixels. The measured data is rounded to the nearest inch and reported as the identity of the landmark.

By matching the the above information with a landmark map, the robot will be able to localize itself the world coordinate frame.

Landmark Recognition Performance The recognition process takes approximately 1.5 seconds on the robot, which runs on an Intel 80386 microprocessor. Measurement revealed that the error in (x, y) was within ± 1 pixels, which translates to ± 0.22 inches in our experimental setup with an 8-foot ceiling. Orientation error was within ± 3 degrees. No landmarks have been misidentified.

For the Next Quarter We proceed to implement the landmark-based navigation using the vision function we have developed this quarter. If necessary, we will attempt to improve the landmarks and/or their recognition algorithm.

4.7 Mobile Robot Navigation Toolkits

The main effort of this quarter has been focused on transferring the toolkit technologies we developed to Nomadic Technologies, a company that designs and manufactures mobile robots for the research and education market. In particular, the following toolkit components have been transferred successfully:

1. The approximate cell decomposition based motion planning module.
2. The artificial potential field based motion control module.
3. The sensor based localization module.
4. The map building module.

Nomadic Technologies is working on further developing these toolkit modules and on integrating them into its Intelligent Robot Software Development Environment.

Application: Using the toolkit, we have demonstrated collaboration between two mobile robots. More precisely, we worked on allowing two mobile robots to reliably recognize each other and to meet at a pre-specified location.

We considered the following scenario: two robots, A and B, are moving within the same space while attempting to complete their separate tasks. At some point in time, robot A decides that it needs to meet with robot B in order to jointly solve some problem, and so sends a request to B that they meet at some location specified relative to a map which both robots share. If able, B then acknowledges that request and both robots move to the rendezvous point.

The main difficultly in completing this task is that we must take into account each robot's positional uncertainty. To do this, each robot attempts to sense the other (using sonar and infrared sensors) as they move together. If one robot senses the other, it adjusts its course appropriately. The robots continue to move until they touch (as determined by pressure-sensitive bumpers). When contact occurs, each robot sends a message to the other to verify that the contact is actually between the two robots, and not between a robot and an obstacle.

After moving together, the robots have a very accurate sense of their positions and orientations relative to each other. This is important if the two robots want to coordinate their motions after making contact with each other.

To perform this experiment, we extended our software to allow general communication between multiple robots. The amount of new software required was minimal. In particular, we re-used code for path planning and control which was part of the existing toolkit. To debug the software for this experiment, we also used the simulator for multiple robots (see below).

4.8 Simulator for Multiple Robots

In this quarter, we continued to work the extending the two-robot simulator to simulate more than two robots. We have finished a prototype implementations of multiple robot simulator that simulate up to 10 robots (a practical limitation rather than a theoretical limitation). We have developed a graphical user interface and a language user interface to allow easy interface and control of multiple robots, directly from the server or from client processes. We have performed extensive experiment with this multiple robot simulator.

Based on this experimentation, we have identified the computational limitation of the centralized multiple robot simulation architecture. We continued to investigate the distributed approach to multiple robot simulation.

4.9 Summary of Main Results Obtained So Far

1. Identification of several axes for distributing path planning software in an on-line architecture.
2. A documented Randomized Path Planner package has been made available to other research institution on the computer network. Several organizations are using it.
3. Implementation of parallel versions of RPP on a Silicon Graphics 4D/240 multiprocessor machine and on a local-area network of UNIX-based workstations.
4. Definition of a new, FFT-based method to compute obstacles in configuration space.
5. Definition and implementation of a new path planning method (the vector-based planner) to generate paths for robots with many degrees of freedom.
6. Integration of several path planners (RPP, vector-based planner with/without potential fields) in a package distributed over a network of UNIX-based workstations.
7. Design and implementation of an optimal-time motion planner for closed-loop kinematic chains.
8. Design and implementation of a randomized three-arm manipulation planner for manipulating an elongated object in a 3D cluttered environment.
9. Design and implementation of a new landmark-based mobile robot planning method. Extension of this planner to deal with controllable uncertainty.
10. Definition of the layout of a software toolkit to efficiently develop new navigation systems. Implementation of several toolkits.
11. Partial development of a powerful multi-mobile-robot simulator to facilitate the development and debugging of programs for multiple interacting mobile robots.

4.10 Status

Our research progresses according to schedule.

Chapter 5

Applications and Technology Transfer

It is not possible to develop generic technology without multiple, specific applications to test and refine the ideas and implementations. As such, we are actively seeking sites, both internally and externally to provide the compelling test beds that will make this project succeed. These driving applications span a variety of the most important target users: high-performance control, intelligent machine systems, underwater vehicle command and control, and remote teleoperation. Several of these projects will reach for new limits in advanced technology and system integration; others will address real-world problems in operational systems.

With the reduced funding levels, we will not have the resources to support all of the originally proposed technology evaluation sites. However, we believe these sites are crucial to the development of ControlShell into a viable technology for "real-world" use. Thus, we have actively pursued alternative means of supporting external sites. We have been successful in securing several new test applications. These sites will either function with minimal support, or fund their own support.

This chapter highlights some of the activities of these projects.

The currently-active ControlShell applications are:

- Precision Machining, by The Stanford Quiet Hydraulics Laboratory.
- Underwater Vehicle Control, a joint project between the ARL and the Monterey Bay Aquarium Research Institute.
- Intelligent Machine Architectures, by Lockheed Missiles and Space Corporation.
- Remote Teleoperation, by Space Systems Loral Corporation.
- Space-based Mobile Robot Systems, by several ARL students (NASA-sponsored).

- High-Performance Control of Flexible Structures, by several ARL students (AFOSR-sponsored).
- Space-structure assembly, by NASA Langley Research Center.
- Mobile-robot control by NASA Ames Research Center.

5.1 NASA sites

Partially as a result of the architectures seminar series, two NASA sites have committed to using ControlShell in their projects. The first site is a robotics laboratory at NASA's Langley Research Center. This laboratory will be studying robotic construction of space structures, using a single industrial robot. The work focuses on end-effector development, system integration, and tools to ease construction planning.

Another NASA lab, the intelligent mechanisms group at NASA Ames, is also planning to utilize ControlShell for a mobile robotics project. This project's goals include studying intelligent exploration algorithms and robotic control architectures.

These sites will fund their own support.

5.2 Transfer of Planning Technology

- Part of our mobile robot software (simple planner, simulation) is being ported by Nomadic Technologies, and is part of the software distributed by this company with their mobile robot NOMAD 200.
- J.C. Latombe and L. Kavraki assisted Nova Management, Inc., in building an automated route planner for tanks in support of US Government Contract No. DAAE07-C-93-0026. A prototype version of this planner was successfully demonstrated to Army representatives.

Publications So Far:

L. Kavraki, *Computation of Configuration-Space Obstacles Using the Fast Fourier Transform*, Technical Report, STAN-CS-92-1425, 1992.

L. Kavraki, *Computation of Configuration-Space Obstacles Using the Fast Fourier Transform, Proc. of the IEEE Int. Conf. on Robotics and Automation*, Atlanta, GA, 1993.

L. Kavraki, J.C. Latombe, and R.H. Wilson, "On the Complexity of Assembly Partitioning," accepted for publication in *Information Processing Letters.*

Y. Koga and J.C. Latombe, "Experiments in Dual-Arm Manipulation Planning," *Proc. of the IEEE Int. Conf. on Robotics and Automation*, Nice, May 1992, pp. 2238–2245.

Y. Koga, T. Lastennet, J.C. Latombe, and T.Y. Li, "Multi-Arm Manipulation Planning," *Proc. of the 9th Int. Symp. on Automation and Robotics in Construction*, Tokyo, June 1992.

J.C. Latombe, "Geometry and Search in Motion Planning," *Annals of Mathematics and Artificial Intelligence*, 8(2-4), 1993.

J.C. Latombe, "Robot Algorithms," *Proc. of the LAAS/CNRS 25th Anniversary Conf.*, Cepadues, Toulouse, France, May 1993, pp. 81–94 (invited conference).

A. Lazanas and J.C. Latombe, *Landmark-Based Robot Navigation*, Rep. No. STAN-CS-92-1428, Dept. of Computer Science, Stanford U., May 1992. Accepted for publication in *Algorithmica.*

A. Lazanas and J.C. Latombe, "Landmark-Based Robot Navigation," *Proc. of the 10th Nat. Conf. on Artificial Intelligence, AAAI-92*, San Jose, July 1992, pp. 816–822.

A. Lazanas and J.C. Latombe, "Landmark-Based Robot Motion Planning," *Proc. of the AAAI Fall Symp.*, Boston, MA. October 1992, pp. 98-103.

A. Lazanas and J.C. Latombe, "Landmark-Based Robot Motion Planning," *Geometric Reasoning for Perception and Action*, C. Laugier (Ed.), Lecture Notes in Computer Science, 708, Springer-Verlag, 1993.

Gerardo Pardo-Castellote and Robert H. Cannon Jr. "Proximate time-optimal parameterization of robot paths," STAN-ARL-92- 88, Stanford University Aerospace Robotics Laboratory, April 1993.

Gerardo Pardo-Castellote, Tsai-Yen Li, Yoshihito Koga, Robert H. Cannon Jr., Jean-Claude Latombe, and Stan Schneider," "Experimental integration of planning in a distributed control system. In *Preprints of the Third International Symposium on Experimantal Robotics*, Kyoto Japan, October 1993.

S. Schneider and R. H. Cannon. "Object impedance control for cooperative manipulation: Theory and experimental results," *IEEE Journal of Robotics and Automation*, 8(3), June 1992. Paper number B90145.

S. A. Schneider and R. H. Cannon. "Experimental object-level strategic control with cooperating manipulators," *The International Journal of Robotics Research*, 12(4):338–350, August 1993.

Howard H. Wang, Richard L. Marks, Stephen M. Rock, and Michael J. Lee. "Task-based control architecture for an untethered, unmanned submersible," In *Proceedings of the 8th Annual Symposium of Unmanned Untethered Submersible Technology*, pages 137–147. Marine Systems Engineering Laboratory, Northeastern University, September 1993.

FSM Editor

Revision 0.9
The ControlShell Finite-State Machine Graphical Editor

1. Introduction

The **FSM Editor** is a graphical tool for creating and viewing *ControlShell* Finite-State Machines. It generates data files that can be directly used in *ControlShell* applications by performing calls to the **`CSFsmParseFile()`** C-function.

The basic graphical components are states, state transitions, and labels. The states are represented by rectangles, the transitions are represented by arrows, and labels are represented by text. An example of a finite-state machine is shown in Figure 1.

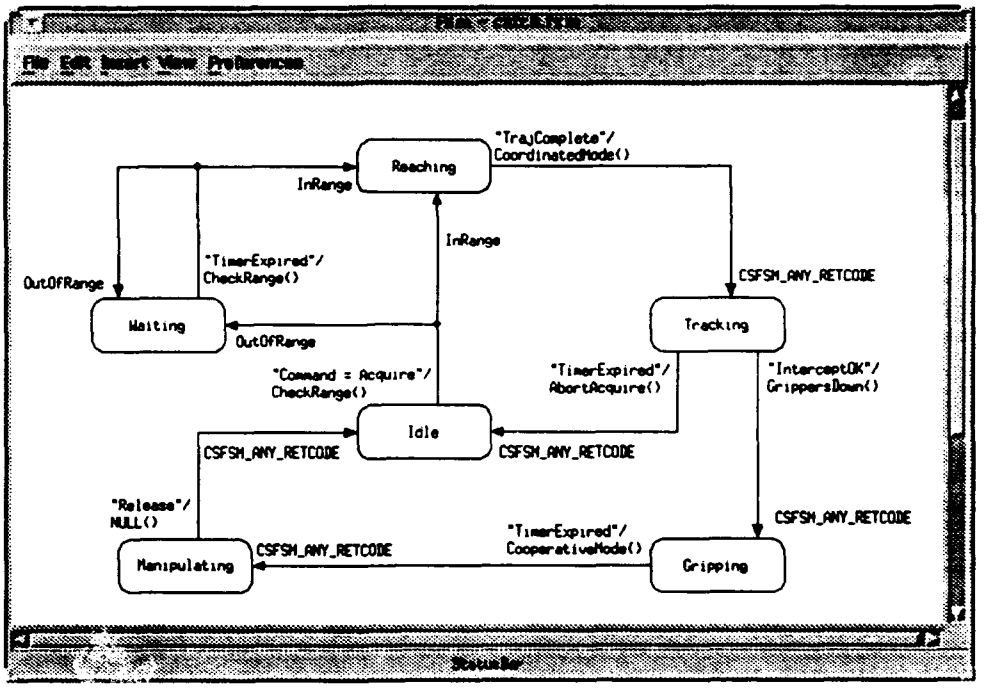


Figure 1 Finite-State Machine Example

Labels can be used to add notes, but are generally bound to states to provide state names, and bound to arrows to provide stimulus names, transition routines and return codes. There are also blocks representing **Subchains**, which are sets of state transitions that may be used in multiple places, and are analogous to subroutines in traditional programming languages.

This document assumes a working knowledge of *ControlShell* FSMs and just describes the steps involved in creating and maintaining finite-state machines.

1.1. Before You Begin

FSM Editor requires a resource file called **Fsm** in the **app-defaults** directory. This file specifies all the menu text, quick-keys and the translations from mouse events to drawing actions. It also specifies resources such as colors and grid size. The default **Fsm** file may be found in the **/local/applications/rti/app-defaults** directory. You should copy it to your personal **~/app-defaults** directory or a global app-defaults directory set up by your system administrator.

Set the **XUSERFILESEARCHPATH** environment variable to make sure the **FSM Editor** can find the resource file. For example, type at the UNIX prompt:

```
setenv XUSERFILESEARCHPATH ~/app-defaults/%N%S
```

Also make sure the **LD_LIBRARY_PATH** includes the **X11R5** directory. Run:

```
setenv | grep LD_LIBRARY_PATH
```

If the result does not include the **X11R5** directory, and your **X11R5** libraries are located in **/local/X11R5/lib/sun4**, type:

```
setenv LD_LIBRARY_PATH /local/X11R5/lib/sun4:$LD_LIBRARY_PATH
```

1.2. Starting the FSM Editor

To start the **FSM Editor**, create and change to a directory that is to hold finite-state machine data files. Type **fsm&** at the UNIX prompt. You will be presented with the initial drawing screen.

2. Creating and Editing a Finite State Machine

The **FSM Editor** allows a finite-state machine to be created with simple mouse actions, and relatively familiar drawing motions. Only the left and middle mouse buttons are used at this time, in conjunction with the **<Shift>** and **<Ctrl>** keys.

The menu bar contains several main menus: **File**, **Edit**, **Insert**, **View**, and **Preferences**. The **File** menu provides access to files. The **Edit** menu provides additional editing commands, such as **Undo**, **Delete**, and **Duplicate**. The **Insert** menu provides a way of inserting Subchains, SubchainEnd, as well as states and labels. The **View** menu provides zoom capability. The **Preference** menu provides commands for toggling snap-to-grid mode and auto-save mode.

The tables in the following subsections further describe mouse actions and the menu options.

2.1. Using the Mouse

This section describes how the mouse is used.

2.1.1. Drawing using the Mouse—Middle Button

The drawing actions associated with the middle mouse button are defined in Table 1.

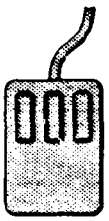


Table 1 Using the Middle Mouse Button

Middle Mouse Button	Action
Click-Drag	Create a transition Arrow. If the starting point is near an arrow head, a new segment is added to the arrow. If the starting point is near another transition line, a new branch is added to the transition.
<Shift>-Click-Drag	Create a state rectangle.
<Ctrl>-Click-Drag	Create a label.

When a transition arrow is created, the default stimulus/transition-routine pair is:

```
CSFSM_ALWAYS_STIM/
NULL()
```

and the default return code is:

```
CSFSM_ANY_RETCODE
```

You must double-click the left mouse button on each label to make changes.

When a state is created, it is assigned a name of **State***n* where *n* is an integer that increments as more states are created.

Similarly, when a label is created, it is assigned the text, **Label***n*.

2.1.1.1. Transition-Arrow Labels

For each transition there is a blue label representing the stimulus/ transition-routine pair and one or more green labels representing return codes, one for each branch of the transition.

The stimulus/transition-routine pair may be entered in a variety of ways, but in general, the stimulus should appear in between double-quotes. If the stimulus is the special keyword, **CSFSM_ALWAYS_STIM**, it should not be in double-quotes. The stimulus text and the transition routine name should be separate by a slash (/) and/or a new line. The parenthesis after the transition routine name is optional, but will be added when you read the data file back into the graphical editor.

Please note that when transitioning into a Subchain or SubchainEnd block, no transition routines are needed (or used).

Some possible representations of stimulus/transition-routine pairs are:

"stim1"/ trans1()	"stim1"/trans1()	"stim1" trans1()	"stim1" trans1

The return-code text should be an integer, a special keyword, **CSFSM_ANY_RETCODE**, or other text. As a convenience, when you save your FSM diagram, the FSM Editor will generate a header file **enum**erating your return code text.

Please note that when the transition is entering a Subchain, the return-code text should be the name of the entry state of the Subchain.

2.1.1.2. State-Name Label

The name of the state can contain any alphanumeric character and underscore (_), but it must be a single word. No white space is allowed.

2.1.1.3. Label Text

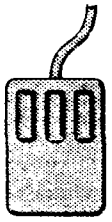
A label text may contain any text and may be any number lines. It is always center aligned.

2.1.2. Editing using the Mouse—Left Button

The actions associated with the left mouse button are defined in Table 2.

Table 2 Using the Left Mouse Button

Left Mouse Button	Action
Click	Deselect all objects, then select the object within range (pick radius) of the mouse pointer.
<Shift>-Click	Toggle the selection of the object within range of the mouse pointer.
Click-Drag	If the mouse pointer is within range of a selected object, all selected objects are moved. If the mouse pointer is near a corner of a state object, resize the state. Otherwise, start a selection rectangle. When the button is released, all objects completely within the selection rectangle will be selected.
Double-Click	If the mouse pointer is within range of a Label object, an edit dialog box is popped up for changing the label text. If the mouse pointer is within range of a Subchain object, a separate edit window for the subchain is activated. Otherwise, no action.



An arrow is composed of "nodes". Generally, each node is where an arrow bends or branches. The user can move individual nodes of an arrow or line segments. To select only a node, first deselect the arrow, then click near one of the nodes; active nodes are represented by small black squares. To select a line segment, click on the segment or <shift>-Click on the two nodes that make up the segment.

If a state object is selected, all arrow segments attached to it will also be selected. *Be careful when deleting after selecting a State!* An arrow is "attached" to a state if its tail node or its arrow-head node is on the bound-

ary of the rectangular state object. Arrow nodes and states snap to a grid (default 20 pixels), easing the task of attaching arrows to states. Label objects do not snap; it makes them easier to place in the drawing window.

The Subchain window, activated when clicking on a Subchain block, is almost identical to the main FSM window. One difference is that a SubchainEnd block is always present. Another is that you cannot quit the program from the **File** menu. You can only choose **Close** to hide the window. Double-clicking on the Subchain block will activate the window again.

When editing a state name, the dialog box will prevent any white space to be inserted, forcing a single word.

2.2. Using the Menus

Using the menus, the user can open and save FSM files, duplicate and delete objects, and insert FSM objects. The menus are separated into five (5) categories, **File**, **Edit**, **Insert**, **View** and **Preference**. The following sections describe the functions of each menu command.

The main FSM drawing window and the Subchain drawing windows contain the same menus, and their operations are the same, except where noted.

If the menu command has a QuickKey combination, it is noted in parentheses after the command name.

2.2.1. File Menu (Meta+F)

The **File** menu performs file operations to save and retrieve FSM files.

2.2.1.1. New (Ctrl+N)

Choosing **File**, **New** will clear the FSM Window, prepared to start a new finite-state machine definition. If the current finite state machine has been edited, the user will be prompted to save it first.

2.2.1.2. Open (Ctrl+O)

Choosing **File**, **Open** will retrieve an existing FSM definition from a file. If the current FSM has been edited, the user will be prompted to save it first. The editor will then pop up the **File**, **Open** selection box, as shown in Figure 2.

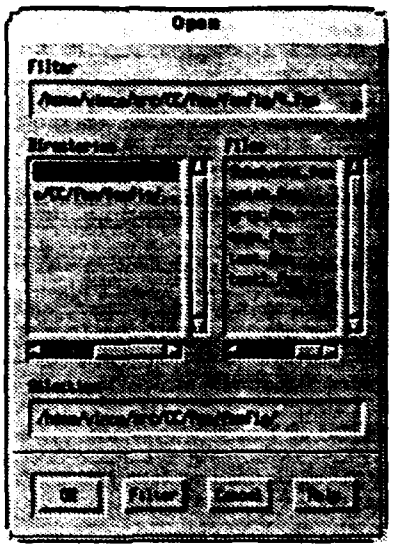


Figure 2 File-Open Selection Box

Use the **Directories** list box to move between directories by double-clicking on the desired directories. To quickly jump to another directory, you can replace the filter string. For example, to jump to the

```
/local/applications/rti/fsm
```

directory to look for FSM files, replace the string in the **Filter** text box with:

```
/local/applications/rti/fsm/*
```

and hit **<Enter>** or click the **Filter** button. Do not forget the "*" wildcard.

To choose a file, double-click on the file name in the **Files** list box, or highlight the filename and click **OK**.

Parsing Errors

The parser checks for errors in the FSM file when it is opened. If there is an error, a dialog pops up to ask if the file should be re-read. This gives the user a chance to edit the file before continuing. The UNIX command window from which **fsm** was invoked will contain the line number at which the error occurred.

Although re-reading will confirm that errors have been removed, there may be extra objects left in the FSM window. It is best to call **File**, **Open** again.

If the FSM file contains Subchains, the Subchain files are not read immediately. A Subchain file is read the first time the user chooses to edit it by double-clicking on the Subchain object.

2.2.1.3. Save (Ctrl+S)

Choosing **File**, **Save** saves the current drawing window. If there are any Subchains that have been edited, they also will be saved. Choosing **File**, **Save** from a Subchain window will not prompt for an FSM name; the **FSM Editor** will use the FSM name from the main FSM window.

If the current window has not been saved before (or was not read from a file), the **File**, **Save As** selection box pops up to ask for a new file name. If the selected file already exists, the user will be asked to confirm overwriting the old file. If any Subchains in the window have not yet been saved, a **File**, **Save As** selection box will pop up for each unsaved Subchain. Look at the caption of the file-selection box to determine which Subchain is being saved.

Backup File

File, **Save** saves a backup file with a file name that contains an additional **.bak** extension.

Header File

To help you write code, **File**, **Save** also generates a header file that enumerates the return codes so you can use descriptive names to describe results of the transition routines. It also defines constant string variables for each unique stimulus expression to minimize errors caused by typos when referring to stimulus expressions in your code.

2.2.1.4. Save As (Ctrl+A)

Choosing **File**, **Save As** saves the current FSM under a new name. If the current window is a main FSM window, it will prompt for an FSM name that is to be registered with *ControlShell* at run-time.

If any Subchains in the window has been edited, it also will be saved. If the Subchain is already tied to a file name, however, **FSM Editor** will *not* prompt for a new file name; you must choose **File**, **Save As** from the Subchain window to change its filename. Moreover, the label text of a Subchain block is updated to reflect the new name.

2.2.1.5. Generate EPS... (Ctrl+E)

File, **Generate EPS** creates an Encasulated PostScript (EPS) file of your FSM diagram. You can easily import this file into your documentation files.

You will be prompted for a file name, and if you leave out the file extension, **.eps** will be appended.

Most document preparation applications will adjust the size of the EPS figure to fit inside the document. If your FSM drawing is large, the figures may be reduced too much to be legible. In these cases, you may want to invoke special scaling and cropping features of your document-preparation application to display portions of the EPS file.

To change the properties of the generated EPS file, such as printer resolution and fonts, make the changes in the **Fsm** app-defaults file.

2.2.1.6. Close

File, **Close** is only available from a Subchain window. It hides the window until it is called forth again by double-clicking on its Subchain object.

2.2.1.7. Quit (Meta+Q)

File, **Quit**, available only from the main FSM window, quits the **FSM Editor** application. If any drawing window has not been saved, the user will be prompted to confirm saving. If, in addition, any drawing window has never been saved, the **File**, **Save As** selection box is popped up to ask for a new file name.

2.2.2. Edit Menu (Meta+E)

The edit menu provides some rudimentary editing commands.

2.2.2.1. Undo (Meta+Bksp)

Choosing **Edit**, **Undo** can undo some commands. If there is an action that can be undone, the **Undo** button will be enabled. Otherwise, the button is "greyed out". Some of the commands that may be undone are:

- Creation of objects (State, Arrow, Label, Subchain, SubchainEnd)
- Deletion of objects (via **Edit**, **Delete**)

You cannot undo moves or file operations.

2.2.2.2. Duplicate (Ctrl+D)

Duplicate is used to duplicate a state, label, or Subchain object. It is especially useful for Subchains, because it ensures that multiple references to the same Subchain in a single drawing window will all point to the same Subchain window.

2.2.2.3. Delete Node

Delete the active line nodes. If the node at the tail end of a transition arrow is selected, the entire transition (all branches) are delete. Use this command to clean up unnecessary nodes in transition arrow.

2.2.2.4. Delete (Bksp)

Choosing **Edit**, **Delete** will delete all selected objects.

Be careful when deleting state objects, since selecting a state also selects all arrows attached to it. To delete a state without deleting the attached arrows, use **<Shift><LeftClick>** to toggle OFF the selection of the arrow nodes.

Additionally, if a return code label is deleted, the arrow (branch) associated with that return code is also deleted. If the stimulus/transition routine label is deleted, the entire transition (all branches) is deleted.

Similarly, if the state-name label is deleted, the state is also deleted.

2.2.3. Insert Menu (Meta+I)

The **Insert** menu provides a way of creating various FSM objects. It always places the new object in the upper-left corner of the drawing window.

2.2.3.1. State (Meta+S)

Choosing **Insert**, **State** creates a new state object. Using this method of creating a state object ensures that all the states are the same size.

2.2.3.2. Label (Meta+L)

Choosing **Insert**, **Label** creates a new label object. Using this may be quicker than using the mouse, and is easier to remember than **<Ctrl><MiddleButtonDrag>**.

2.2.3.3. Subchain (Meta+C)

Choosing **Insert**, **Subchain** is the only way to create a new Subchain object. If you need to refer to an existing Subchain, use **Edit**, **Duplicate**. Double-click on the subchain to activate the Subchain window for editing. The file name represents the name of the Subchain object. You cannot directly change the name of a Subchain by editing its label.

Please see the next chapter on descriptions of transitions to and from Subchain objects.

2.2.3.4. SubchainEnd (Meta+E)

Choosing **Insert**, **SubchainEnd** is the only way to create a new SubchainEnd object. The label must be an integer and represents the return code from the Subchain. A Subchain window may have multiple SubchainEnd objects representing multiple return codes.

Please see the next chapter on descriptions of transitions to and from SubchainEnd objects.

2.2.4. View Menu (Meta+V)

The **View** menu contains zoom factors for displaying the drawing: 25%, 50%, 75%, 100%, 125%, 150%, 200%. In the current version, the text in the FSM Editor is not scaled, so positioning will appear correctly only at 100% zoom. Below 75% zoom, all text will be represented by a single period (.).

2.2.5. Preferences Menu (Meta+P)

The **Preferences** menu allows the setting of user preferences for the FSM Editor.

2.2.5.1. Snap-to-Grid Mode (Ctrl+G)

Toggle the snap-to-grid behavior when drawing and placing states and transition arrow. The grid size cannot be set from within the editor. You must set the associated resource in the **app-defaults** file or in **.Xdefaults**:

```
Fsm*Stage.grid:   20
```

2.2.5.2. Auto-Save Mode

Toggle auto-saving of the current FSM drawing. The auto-save interval is specified in the **app-defaults** file in units of minutes:

```
Fsm*autosaveInterval:    2
```

The default interval is 5 minutes.

The autosave file has the same name as the current file, with an appended '#' character. Thus, the auto-save file for **main.fsm** is **main.fsm#**. After the file is explicitly saved the auto-save file is deleted.

If the current file has never been saved, Auto-Save will pop up the **File, Save As** file-selection box to prompt for a file name.

3. Finite State Machine Objects

The finite state machine (FSM) objects in the drawing window of **FSM Editor** represent different parts of the FSM definition. Each object has some special properties associated with it, including its interaction with other objects. The interconnections amongst the objects are important to creating a viable finite state machine that can be used by *ControlShell*.

The **FSM Editor** will allow you to save and retrieve partially completed FSM files, letting you work at your own pace, but *ControlShell* will not be able to successfully parse these incomplete FSM files at run-time. When you save an FSM, the **FSM Editor** will warn you of the inconsistencies.

The subsections that follow describe the objects in more detail.

3.1. State

A state is represented by a rectangle on the drawing screen, with an associated label giving the state name. The special keyword, **CSFSM_ANY_STATE** can be used to represent the "Global" state, defining responses to stimuli that should occur from anywhere in the FSM.

Each state should have at least one transition from it, represented by the tail of a transition arrow attached to its border, and one transition to it, represented by the head of a transition arrow attached to its border. Each transition *from* the state must have a unique stimulus/transition-routine pair, i.e., there is only one way for each state to respond to a particular stimulus.

All return codes of transitions *to* a state must be an integer, or the special keyword, **CSFSM_ANY_RETCODE**.

When a state is selected by the left mouse button, all transition tail and head nodes are also selected. This allows you to move a state and still retain its connections. If you do not want to move the nodes, **<Shift><LeftClick>** each node to toggle OFF its selection.

3.2. Transition Arrow

A transition arrow drawing object represents many elements of a FSM definition. The label at the tail of an arrow contains both the stimulus to which

a state is to respond and the transition routine that is to be executed upon receipt to that stimulus. The transition routine may return different return codes, and each branch of an arrow represents the possibilities, along with the next state corresponding to each return code.

3.2.1. Stimulus/Transition Routine Pair

The stimulus/transition-routine pair may be entered in a variety of ways, but in general, the stimulus should appear between double-quotes. If the stimulus is the special keyword, **CSFSM_ALWAYS_STIM**, it should not be in double-quotes. The stimulus text and the transition routine name should be separate by a slash (/) and/or a new line. The parenthesis after the transition routine name is optional, but will be added when you read the data file back into the **FSM Editor**.

Some possible representations of stimulus/transition-routine pairs are:

"stim1"/ trans1()	"stim1"/trans1()	"stim1" trans1()	"stim1" trans1

3.2.2. Return Code

The return code can be an integer, the **CSFSM_ANY_RETCODE** keyword, or text that describe the return condition. If descriptive text is used, they are enumerated in a header file using **enum** to help you write your C or C++ code. The first **enum** value starts at **100**, allowing to use explicit integers from **0** to **99**.

3.2.3. Special Cases

There are special rules for transition arrows when transitioning to/from Subchain and SubchainEnd objects.

3.2.3.1. Subchain

There can be only *one* transition *to* each Subchain object. If the Subchain can be used in another place, use **Edit**, **Duplicate** to create another Subchain object in the drawing window.

The transition to a Subchain *does not have* a transition routine, which means that the transition cannot branch. Any transition routine listed will be ignored. The "return code" of the arrow entering the Subchain *must* the name of a state in the Subchain. This represents the initial state of the Subchain.

There can be only *one* transition *from* each Subchain object, but the transition may branch, one for each return code. The transition has *no* stimulus or transition routine name.

3.2.3.2. SubchainEnd

There can be only multiple transition *to* each SubchainEnd object, but for each transition, there are no transition routines, which means none of the arrows may branch.

There are *no* return codes for transitions *to* a SubchainEnd object.

There are *no* transitions *from* a SubchainEnd object.

3.3. Subchain

A subchain represents a reusable FSM module. It contains a series of state transitions that may be utilized in many places. It is analogous to a subroutine or function in traditional programming languages. Each Subchain object is actually a gateway to a Subchain drawing window.

Subchain drawing windows are used just like the main drawing window and may contain more Subchain objects. For each Subchain drawing window, there must be at least one SubchainEnd object to indicate a return from the Subchain to the calling FSM level.

3.4. SubchainEnd

A SubchainEnd object represents an exit point from a Subchain. Its label represents the return code that the subchain will return to the calling FSM. It is analogous to the **`return`** statement in C.

Multiple SubchainEnd objects may appear in a Subchain window, each representing a different return code.

3.5. Label

A label object is just a place where you can add text or comments. It can be multi-lined, but is always center aligned.

4. Finite State Machine Data Files

The data file used by the **FSM Editor** is the same as that used by the *ControlShell* run-time parser. The only additions are coordinates for the drawing objects.

Figure 3 and Figure 4 show the graphical representation of the FSM example shipped with *ControlShell*.

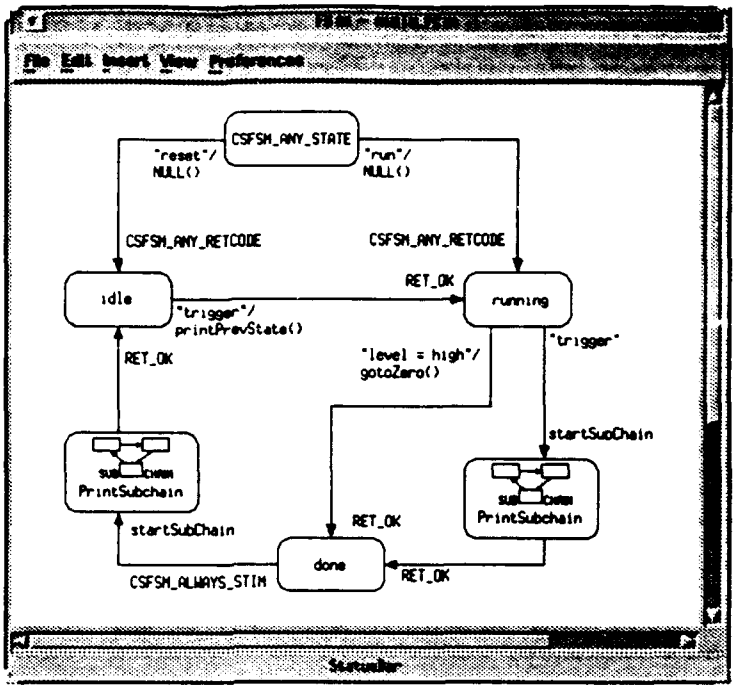


Figure 3 Main FSM Example

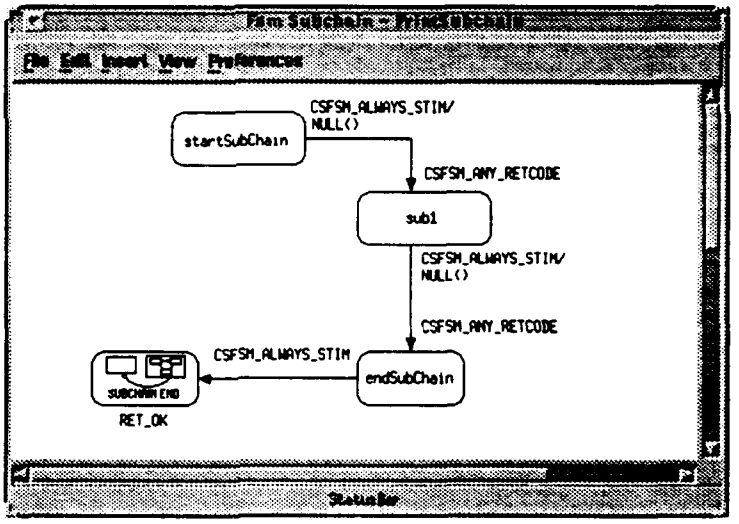


Figure 4 Subchain FSM Example

Figure 5, Figure 6, and Figure 7 show the corresponding data files[1].

```
# /home/vince/src/CC/fsm/fsmfig main.fsm
#
# Generated by fsm, Fri Dec  3 18:59:39 1993


CSFsm: plan done        [200,340,80,40][240,360]        65
    "CSFSM_ALWAYS_STIM"         CSFsmSubchainStart      startSubChain
[90,380] [200,360;80,360;80,320] [90,340] [40,260,100,60] [90,298]
/home/vince/src/CC/fsm/fsmfig/PrintSubchain
        101 -> idle      [80,260;80,180] [86,211]

CSFsm: plan running     [340,140,80,40][383,160]        22
    "level = high"      gotoZero        [263,221]
        101 -> done      [360,180;360,240;240,240;240,340] [257,334]

CSFsm: plan running     [340,140,80,40][383,160]        22
    "trigger"   CSFsmSubchainStart      startSubChain     [404,197]
[400,180;400,280] [404,267] [340,280,100,60] [390,318]
/home/vince/src/CC/fsm/fsmfig/PrintSubchain
        101 -> done      [400,340;400,360;280,360] [294,375]

CSFsm: plan idle        [40,140,80,40][79,159] 14
    "trigger"   printPrevState      [124,189]
        101 -> running      [120,160;340,160] [297,153]

CSFsm: plan CSFSM_ANY_STATE     [160,20,100,40][211,40]         0
    "run"       NULL        [265,71]
        CSFSM_ANY_RETCODE -> running      [260,40;380,40;380,140] [270,122]

CSFsm: plan CSFSM_ANY_STATE     [160,20,100,40][211,40]         0
    "reset"     NULL        [107,71]
        CSFSM_ANY_RETCODE -> idle      [160,40;80,40;80,140] [87,123]

Retcode: RET_OK101
```

Figure 5 Main FSM Example Data File

[1]The data formats are subject to change pending final release of *ControlShell*

```
# /home/vince/src/CC/fsm/fsmfig/PrintSubchain
#
# Generated by fsm, Fri Dec  3 18:59:39 1993


CSFsm: plan endSubChain        [240,200,80,40][280,220]       306
    CSFSM_ALWAYS_STIM  CSFsmSubchainEnd       101     [133,209]
[240,220;120,220] [40,200,80,40] [80,245]

CSFsm: plan sub1       [240,80,100,40][289,99]        298
    "CSFSM_ALWAYS_STIM"        NULL      [289,150]
       CSFSM_ANY_RETCODE -> endSubChain      [280,120;280,200] [289,188]

CSFsm: plan startSubChain      [100,20,100,40][149,41]        290
    "CSFSM_ALWAYS_STIM"        NULL      [206,36]
       CSFSM_ANY_RETCODE -> sub1      [200,40;280,40;280,80] [291,73]
```

Figure 6 Subchain Example Data File

```
/* Name: /home/vince/src/CC/fsm/fsmfig/main.fsm.h

   Generated by fsm, Fri Dec  3 18:59:40 1993

 */

#ifndef _main_fsm_h
#define _main_fsm_h

/* Return codes */

typedef enum {
    MIN_RETCODE = 100,
    RET_OK,
    MAX_RETCODES
} planFsmReturnCode;

/* Stimulus expressions */
#ifndef DEF_CSSTIM_LEVEL_EQ_HIGH
#define DEF_CSSTIM_LEVEL_EQ_HIGH
const char CSSTIM_LEVEL_EQ_HIGH[] = "level = high";
#endif /* DEF_CSSTIM_LEVEL_EQ_HIGH */

#ifndef DEF_CSSTIM_RESET
#define DEF_CSSTIM_RESET
const char CSSTIM_RESET[] = "reset";
#endif /* DEF_CSSTIM_RESET */

#ifndef DEF_CSSTIM_RUN
#define DEF_CSSTIM_RUN
const char CSSTIM_RUN[] = "run";
#endif /* DEF_CSSTIM_RUN */

#ifndef DEF_CSSTIM_TRIGGER
#define DEF_CSSTIM_TRIGGER
const char CSSTIM_TRIGGER[] = "trigger";
#endif /* DEF_CSSTIM_TRIGGER */


#endif /* _main_fsm_h */
```

Figure 7 Main FSM Header File